Translating Hardware Process Algebras into Standard Process Algebras : Illustration with CHP and LOTOS

Gwen Salaün, Wendelin Serwe

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS

Gwen Salaün — Wendelin Serwe

N° 5666

September 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5673--FR+ENG

Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS

Gwen Salaün*† , Wendelin Serwe*‡

Thème COM — Systèmes communicants

Projet VASY

Rapport de recherche n° 5666 — September 2005 — 25 pages

Abstract: A natural approach for the description of asynchronous hardware designs are hardware process algebras, such as Martin's CHP (*Communicating Hardware Processes*), TANGRAM, or BALSA, which are extensions of standard process algebras with particular operators exploiting the implementation of synchronisation using handshake protocols.

In this report, we give a structural operational semantics for value-passing CHP. Compared to existing semantics of CHP defined by translation into Petri nets, our semantics handles value-passing CHP with communication channels open to the environment and is independent of any particular (2- or 4-phase) handshake protocol used for circuit implementation.

In a second step, we describe the translation of CHP into the standard process algebra LOTOS, in order to allow the application of the CADP verification toolbox to asynchronous hardware designs. A prototype translator from CHP to LOTOS has been successfully used for the compositional verification of the control part of an asynchronous circuit implementing the DES (*Data Encryption Standard*).

Key-words: Formal Method, Process Algebra, Asynchronous Hardware Design, Structural Operational Semantics, CHP, LOTOS.

A short version of this research report is also available as "*Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS*", in Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, Proceedings of the Fifth International Conference on Integrated Formal Methods IFM 2005 (Eindhoven, The Netherlands), November 29 – December 2, 2005.

* INRIA Rhône-Alpes
† E-mail: Gwen.Salaun@inria.fr
‡ E-mail: Wendelin.Serwe@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Traduction d'algèbres de processus pour le matériel vers des algèbres de processus standards — Illustration avec CHP et LOTOS

Résumé : Une approche naturelle pour la description d'architectures matérielles et de circuits asynchrones est d'utiliser les algèbres de processus, telles que CHP (*Communicating Hardware Processes* proposée par A. J. Martin), TANGRAM ou BALSA, qui étendent les algèbres de processus standards par des opérateurs particuliers qui exploitent la manière dont la synchronisation est implantée par des protocoles de rendez-vous ("poignée de main").

Dans ce rapport, nous donnons une sémantique opérationnelle structurelle pour CHP avec passage de valeurs. Contrairement à la sémantique existante de CHP définie par traduction vers des réseaux de Petri, notre sémantique traite le cas de CHP avec passage de valeurs et avec des canaux de communication ouverts sur l'environnement. De plus, elle est indépendante de tout protocole particulier de rendez-vous (2 ou 4 phases) utilisé dans l'implantation de circuits.

Dans une seconde étape, nous décrivons la traduction de CHP vers l'algèbre de processus standard LOTOS, afin de permettre l'utilisation de la boîte à outils CADP pour la vérification de conceptions matérielles asynchrones. Un prototype de traducteur CHP en LOTOS a été utilisé avec succès pour la vérification compositionnelle de la partie contrôle d'un circuit asynchrone implantant le standard d'encryption de données DES (*Data Encryption Standard*).

Mots-clés : Méthode formelle, algèbre de processus, conception de circuits asynchrones, sémantique opérationnelle structurelle, CHP, LOTOS.

1 Introduction

In the currently predominating synchronous approach to hardware design, a global clock is used to synchronise all parts of the designed circuit. This method has the drawback that the global clock requires significant chip space and power. Asynchronous design methodologies [Hau95] abandon the notion of global clock: the different parts of an asynchronous circuit evolve concurrently at different speeds, with no constraints on communication delays. The advantages of asynchronous design include reduced power consumption, enhanced modularity, and increased performance. However, asynchronous design raises problems that do not exist in the synchronous approach, e.g. proving the absence of deadlocks in a circuit. Furthermore, an established asynchronous design methodology with industrial tool support is still lacking.

Adequate description languages are necessary to master the design of asynchronous circuits. Several process algebras dedicated to the description of asynchronous hardware have been proposed, as for instance CHP (*Communicating Hardware Processes*) [Mar86], TANGRAM [KP01], or BALSA [EB02], which allow the description of concurrent processes communicating via handshake synchronisations. In these languages, there is no global clock, but each process may have its own local clock as in GALS (*Globally Asynchronous, Locally Synchronous*) architectures. The global control flow results from processes waiting for their partner to engage in a handshake communication. These *hardware* process algebras are based on similar principles as *standard* process algebras [BPS01, Fok00] (such as ACP, CCS, CSP, LOTOS, μCRL, etc.). Especially, they provide operators such as nondeterministic choice, sequential and parallel composition. However, compared to standard process algebras, they offer extensions that capture the low-level aspects of hardware communications. In particular, communication in CHP, TANGRAM, or BALSA is not necessarily atomic (as it is in standard process algebras), and may combine message-passing with shared memory communication. For instance, the *probe* operator [Mar85] of CHP allows to check if the communication partner is ready for a communication, but without performing the communication actually.

CHP, TANGRAM, and BALSA are supported by synthesis tools that can generate the implementation of a circuit from its process algebraic description. For instance, the TAST tool [Ren05] can generate netlists from CHP descriptions and is being used to design complex circuits, e.g. by STMicroelectronics, France Telecom R&D, and the asynchronous hardware group of the CEA/Leti laboratory [BCV+05]. Our goal is to enable the verification of asynchronous hardware designs with CADP [GLM02b], a toolbox for verifying LOTOS [ISO89] specifications.

In this report, we give an SOS (*Structural Operational Semantics*) [BPS01, chapter 3] semantics for value-passing CHP. Compared to the most recent semantics [RY04] for CHP, which is defined by translation into Petri nets, our semantics handles value-passing CHP with communication channels open to the environment and is independent of any particular (2- or 4-phase) handshake protocol (cf. Section 2.2) used for circuit implementation. We present in a second step the principles of a translation from CHP into LOTOS. A prototype translator has been implemented and successfully used for the compositional verification of an asynchronous implementation of the DES (*Data Encryption Standard*) [NIS99].

As regards related work, we notice that the semantics of hardware process algebras is usually not given in terms of SOS rules (as it is the case for standard process algebras), but rather by means of a translation into another formalism, as for instance handshake circuits for TANGRAM [vB93], Petri nets for CHP [RY04], and CSP for BALSA [WKTZ04]; in that

respect, we believe that our SOS semantics is an original contribution. As regards verification of asynchronous circuits described using process algebra, there is very little related work. [BBM+03] proposes a translation of CHP into networks of communicating automata. Contrary to our approach, [BBM+03] can only handle CHP processes with intertwined sequential and parallel compositions by flattening parallel processes, which is less efficient than the compositional approach presented in this report. [WKTZ04] sketches, but does not detail, a translation of BALSA into CSP. A major difference between [WKTZ04] and our approach is that [WKTZ04] cannot translate a BALSA process B independently of the BALSA processes communicating with B, whereas our approach is generic in the sense that each CHP process is translated into LOTOS regardless of its context.

The remainder of the reportis organised as follows. Section 2 presents CHP and highlights its probe operator. An SOS semantics for CHP is given in Section 3, and compared to the Petri net based semantics given for CHP in [RY04]. Section 4 presents translation rules from CHP into LOTOS, and reports on an experiment with a prototype translator. Finally, Section 5 gives some concluding remarks.

2 Main Principles of CHP

In this section, we focus on the behavioural part of CHP and omit additional structures such as modules or component libraries present in the full CHP [Ren05].

2.1 Syntax

A CHP *description* is a tuple $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_1 \parallel \cdots \parallel \hat{B}_n \rangle$ consisting of a finite set of *channels* $\mathcal{C} = \{c_1, \ldots, c_n\}$ for handshake communication, a finite set of *variables* $\mathcal{X} = \{x_1, \ldots, x_n\}$ and a finite set of concurrent *processes* $\hat{B}_i$ communicating by the channels. Without loss of generality, we suppose that all identifiers (channels and variables) are distinct — this can be achieved by using a suitable renaming.

A channel c is either binary (between two processes) or unary (between a process and the environment); in the latter case, c is also called a *port*; the predicate $port(c)$ holds iff c is a port. Channels are unidirectional, i.e. a process can use a channel either for *emissions* or for *receptions*. We write $emitter(i, c)$ (respectively $receiver(i, c)$) if process $\hat{B}_i$ uses channel c for emissions (respectively receptions). Also, a process is either *active* or *passive* for a given channel. We write $active(i, c)$ (respectively $passive(i, c)$) if process $\hat{B}_i$ is active (respectively passive) for channel c. This distinction between active and passive is also present in other hardware process algebras such as TANGRAM and BALSA. Note that $passive(i, c)$ is not the negation of $active(i, c)$, since for a process $\hat{B}_i$ not using c both $active(i, c)$ and $passive(i, c)$ do not hold. Binary channels can only connect matching processes, i.e. for each binary channel, there is one emitter and one receiver, as well as an active and a passive process, both notions being orthogonal. Let $\mathcal{C}_i$ be the set of channels used by $\hat{B}_i$.

We have for any channel c the following properties:

$$(\exists i \in \{1,\ldots,n\})\ \mathit{emitter}(i,c)\ \vee\ \mathit{receiver}(i,c)$$
$$(\exists i \in \{1,\ldots,n\})\ \mathit{active}(i,c)\ \vee\ \mathit{passive}(i,c)$$
$$(\exists i \in \{1,\ldots,n\})\ \mathit{emitter}(i,c) \Longrightarrow \big((\forall j \in \{1,\ldots,n\})\ j \neq i \Longrightarrow \neg \mathit{emitter}(j,c)\big)$$
$$(\exists i \in \{1,\ldots,n\})\ \mathit{receiver}(i,c) \Longrightarrow \big((\forall j \in \{1,\ldots,n\})\ j \neq i \Longrightarrow \neg \mathit{receiver}(j,c)\big)$$
$$(\exists i \in \{1,\ldots,n\})\ \mathit{active}(i,c) \Longrightarrow \big((\forall j \in \{1,\ldots,n\})\ j \neq i \Longrightarrow \neg \mathit{active}(j,c)\big)$$
$$(\exists i \in \{1,\ldots,n\})\ \mathit{passive}(i,c) \Longrightarrow \big((\forall j \in \{1,\ldots,n\})\ j \neq i \Longrightarrow \neg \mathit{passive}(j,c)\big)$$

Using these notations, the predicate $\mathit{port}(c)$ can be defined formally for any channel c as follows:

$$\mathit{port}(c) \Longleftrightarrow \left(\begin{array}{l} \Big((\exists i \in \{1,\ldots,n\})\ \mathit{active}(i,c)\ \wedge\ \big((\forall j \in \{1,\ldots,n\})\ \neg \mathit{passive}(j,c)\big)\Big) \vee \\ \Big((\exists i \in \{1,\ldots,n\})\ \mathit{passive}(i,c)\ \wedge\ \big((\forall j \in \{1,\ldots,n\})\ \neg \mathit{active}(j,c)\big)\Big) \end{array} \right)$$

Each variable is local to a single process, i.e. the set $\mathcal{X}$ is the disjoint union of n sets $\mathcal{X}_1$, …, $\mathcal{X}_n$, each $\mathcal{X}_i$ containing the local variables of process $\hat{B}_i$. There are no shared variables between processes. We suppose the existence of a set of predefined data types (natural numbers, Booleans and bit vectors) with side-effect-free *operations*, written f_1, …, f_n. Variables and channels are typed; the type of a variable x (respectively a channel c) is written as $\mathit{type}(x)$ (respectively $\mathit{type}(c)$). $\mathcal{V}$ stands for the set of value expressions built using the predefined data types and operations.

The behaviour of a process B[1] is described using assignments, communication actions, collateral and sequential compositions, and nondeterministic guarded commands, according to the following grammar:

$B ::=$ `nil`	*deadlock*[2]
$\mid$ `skip`	*null action*
$\mid$ x`:=`V	*assignment*
$\mid$ c`!`V	*emission on channel* c
$\mid$ c`?`x	*reception on channel* c
$\mid$ B_1`;`B_2	*sequential composition*
$\mid$ B_1`,`B_2	*collateral composition*
$\mid$ `@[`$G_1 \Rightarrow B_1$`;` T_1 … $G_n \Rightarrow B_n$`;` T_n`]`	*guarded commands*
$G ::= V$	*Boolean value expression*
$\mid$ c`#`V $\mid$ c`#`	*probe on passive channel*
$T ::=$ `break` $\mid$ `loop`	*terminations*
$V ::= x \mid f(V_1,\ldots,V_n)$	*value expression*

Collateral composition has higher priority than sequential composition, but brackets can be used to express the desired behaviour, e.g. "B_1`,` $(B_2$`;`$B_3)$".

[1] B stands for any process, whereas $\hat{B}_i$ is one of the n processes of the CHP description.

[2] The deadlocking process `nil` is not present in the version of CHP implemented in TAST [Ren05], but is required for the definition of the SOS semantics.

The collateral composition "`,`" and the parallel composition of processes "$\|$" correspond to two different notions of concurrency. The parallel composition "$\|$" specifies concurrent execution with handshake communications between processes, whereas collateral composition "`,`" specifies concurrent execution without any communication, neither by handshakes nor by variables. The following constraints hold for a process "B_1`,`B_2": if B_1 modifies a variable x, B_2 must neither access the value of x nor modify x, and the sets of channels used by B_1 and B_2 must be disjoint (which also prohibits two interleaved emissions or receptions on a same channel).

As regards guarded commands, the guards are either Boolean value expressions or probes on channels for which the process is passive. The keyword `break` indicates that the execution of the guarded command terminates, whereas `loop` indicates that "`@[`$G_1 \Rightarrow B_1$`;` T_1 ... $G_n \Rightarrow B_n$`;` T_n`]`" must be executed once more, thus allowing loops to be specified in CHP. The version of CHP implemented in TAST [Ren05] also allows deterministic guarded commands which are a particular case of nondeterministic guarded commands with mutually exclusive guards. Therefore we consider only nondeterministic guarded commands in this report.

2.2 Informal Semantics of Handshake Communication in CHP

Communication between concurrent processes in CHP is implemented by means of hardware handshake protocols. There exists different protocols, as for instance the 2-phase protocol (based on transition signalling) and the 4-phase protocol (based on level signalling) [RY04]. Each CHP channel c is translated into a set of wires x_c needed to carry data transmitted on c and two control wires c_{req} and c_{gr} implementing an access protocol to x_c. The 2-phase protocol for communication between two processes B_1 (active) and B_2 (passive) on a channel c consists of the following phases:

1. *Initiation.* B_1 sends a *request* to B_2 by performing an electronic transition ("zero-to-one" or "one-to-zero") on c_{req}.
2. *Completion.* B_2 sends an acknowledgement (or *grant*) to B_1 by performing an electronic transition on c_{gr} and the emitted value is assigned to the variable of the receiver, using the wires x_c.

In a 4-phase protocol, sending a request (respectively acknowledgement) is implemented by a value of 1 on wire c_{req} (respectively c_{gr}). After two phases similar to a 2-phase protocol, two additional phases implement the *return-to-zero*, first of c_{req}, then of c_{gr}. Common to both protocols is that a communication on a channel c is initiated by the process active for c, which is blocked as long as the communication has not been completed by the passive process.

The probe operation of CHP was introduced in [Mar85] and has been found to be useful in the design of asynchronous hardware. The notation "c`#`" allows a passive process (either emitter or receiver) to check if its active partner has already initiated a communication on c. The notation "c`#`V", which can only be used by a receiver, checks if the sender has initiated the emission of the particular value V. Contrary to classical process algebra operators, the probe allows a process to obtain information about the current internal state (communication initiated or not) of a concurrent process without performing the communication actually. Typically, probes are used for multiple reads, executing "c`#`V" several times, which avoids the introduction of an additional variable to store V.

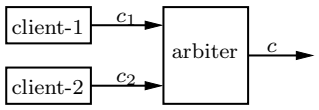


Figure 1: Architecture of the asynchronous arbiter

Similar operators are also present in other hardware process algebras. For instance, Balsa provides a particular form of reception, called *input enclosure* [EB02], that allows the receiver to perform several commands before acknowledging the reception, whereas the sender witnesses an atomic communication.

2.3 Running Example: An Asynchronous Arbiter

Throughout this report we consider the example of an asynchronous arbiter presented in [RY04], which we generalise in two ways: we use value-passing communications instead of pure synchronisations and we model an arbiter open to its environment by keeping the shared resource outside of the arbiter example itself.

Arbiters are commonplace in digital systems, wherever a restricted number of resources (such as memories, ports, buses, etc.) have to be allocated to different client processes. Figure 1 depicts the situation where two clients compete for accessing a common resource. Each client transmits a request for the resource to the arbiter via an individual channel (c_1 or c_2). A third channel allows the arbiter to send the number of the selected client (1 or 2) to the environment, i.e. the resource. The arbiter chooses nondeterministically between the clients with pending requests. The corresponding Chp description is $\langle \{c, c_1, c_2\},\ \{x\},\ \text{client-1} \parallel \text{client-2} \parallel \text{arbiter} \rangle$, where all three channels have an active emitter and a passive receiver, where variable x — taking values in the set $\{1, 2\}$ — is local to the arbiter, and where the three processes are described as follows:

client-1: `@[`*true* $\Rightarrow$ c_1`!1; loop]`
client-2: `@[`*true* $\Rightarrow$ c_2`!2; loop]`
arbiter: `@[`c_1 `#1` $\Rightarrow$ (c`!1,` c_1`?`x`); loop  `c_2 `#2` $\Rightarrow$ (c`!2,` c_2`?`x`); loop]`

In this example, the arbiter uses the probe operator to check if a client has a pending request for the resource.

3 A Structural Operational Semantics for CHP

In this section, we give an SOS semantics for Chp with value-passing communications. Contrary to the existing semantics for Chp [Mar86, RY04], we define the semantics of Chp without expanding communications according to a particular handshake protocol. Thus, our approach is general in the sense that it gives to any Chp description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_1 \parallel \cdots \parallel \hat{B}_n \rangle$ a unique behavioural semantics by means of an Lts (*Labelled Transition System*). In this Lts, a state contains two parts (data and behaviour); a transition corresponds either to an observable action (communication on a channel)[3] or an internal action, written τ, of a process. Following [RY04], internal actions are generated whenever a process assigns one of

[3] Chp has no hiding or restriction operator; thus all inputs and outputs are observable.

its local variables. Our definitions adopt the usual interleaving semantics of process algebras, i.e. at every instant, at most one observable or internal action can take place.

We first present the notion of environment describing the data part of our semantics. Then, we define the behavioural part in two steps, starting with the semantics of a single process evolving independently, followed by the semantics of a set of communicating processes $\hat{B}_1 \parallel \cdots \parallel \hat{B}_n$. Finally, we compare our semantics with the two semantics of [RY04] for 2- and 4-phase handshake protocols.

3.1 Environments

The main semantic difficulty in CHP is the handling of the probe, since this operator exploits the fact that communication is not atomic at the lower level of implementation. Inspired by the actual hardware implementation of CHP, we associate to each channel c a variable noted x_c that is modified only by the process active for c, but might be read by the process passive for c. For a channel c with an active emitter, the type of x_c is $type(c)$; the active emitter assigns the emitted value to x_c when initiating the communication, and resets x_c (to the undefined value, written $\perp$) when completing the communication. A variable x_c is equal to $\perp$ iff all initiated communications on c have been completed. For a channel with an active receiver, the type of x_c is the singleton $\{ready\}$ representing that the active receiver has initiated the communication. Formally, we define the extended set of variables as $\mathcal{X}^* = \mathcal{X} \cup \{x_c \mid c \in \mathcal{C}\}$, and define $\mathcal{X}_i^*$ as the set of the local variables of $\hat{B}_i$ and all the variables x_c such that channel c is used by $\hat{B}_i$. Notice that the additional variables x_c allow to ensure that a value sent by the active process on channel c can be read — or probed — as often as desired by the passive process before completion of the communication.

We define an environment E on $\mathcal{X}^*$ as a partial mapping from $\mathcal{X}^*$ to ground values (i.e. constants), and write environments as sets of associations $x \mapsto v$ of a ground value v to a variable x. Environment updates are described by the operator $\oslash$, which is defined as follows:

$$(\forall x \in \mathcal{X}^*)\ \big(E_1 \oslash E_2\big)(x) = \begin{cases} E_1(x) & \text{if } E_2(x) = \perp \\ E_2(x) & \text{otherwise} \end{cases}$$

The environment obtained by resetting a variable x to $\perp$ in an environment E is described by the function $reset(E, x)$.

The semantics of a value expression V is defined by the usual evaluation function $eval(E, V)$ extended for the probe operator:

$$\begin{array}{l} eval(E, x) = E(x) \\ eval\big(E, f(V_1, \ldots, V_n)\big) = f\big(eval(E, V_1), \ldots, eval(E, V_n)\big) \\ eval(E, C\,\#) = true \iff E(x_c) = ready \\ eval(E, C\,\#V) = true \iff E(x_c) = eval(E, V) \end{array}$$

3.2 Behavioural Semantics for a Single Process

Our semantics associates to each process $\hat{B}_i$ an LTS $\langle \mathcal{S}_i, \mathcal{L}_i, \rightarrow_i, \langle E_i, \hat{B}_i \rangle \rangle$, where

- The set of states $\mathcal{S}_i$ contains pairs $\langle E, B \rangle$ of a process B and an environment E on $\mathcal{X}_i^*$.

- The set of labels $\mathcal{L}_i$ contains emissions, receptions, τ (representing assignments to local variables), and a particular label $\surd$ representing successful termination.
- The transition relation "$\rightarrow_i$" is defined below by SOS rules similar to those used for BPA_ε (*Basic Process Algebra with* ε) in [BPS01, chapter 3]; as for BPA_ε, we write $\langle E, B\rangle_\surd$ to mean $\langle E, B\rangle \xrightarrow{\surd}_i \langle E, \texttt{nil}\rangle$.
- The initial state is $\langle E_i, \hat{B}_i\rangle$, where the initial environment E_i assigns the undefined value $\perp$ to all variables of $\mathcal{X}_i^*$.

First of all, there are no rules for `nil`.

Rules for `skip`. The process `skip` always terminates successfully.

$$\frac{}{\langle E, \texttt{skip}\rangle_\surd}$$

Rules for Assignments. An assignment can always be executed and modifies the environment by updating the value associated to the assigned variable:

$$\frac{}{\big\langle E,\ x\texttt{:=}V\big\rangle \xrightarrow{\tau}_i \big\langle E \oslash \{x \mapsto \mathit{eval}(E,V)\},\ \texttt{skip}\big\rangle}$$

Rules for Emissions. A passive emission is always possible. An active emission on a channel c involves two transitions: the first one assigns a value to x_c and the second one completes the communication by resetting x_c.

$$\frac{\mathit{passive}(i,c)}{\big\langle E,\ c\texttt{!}V\big\rangle \xrightarrow{c\texttt{!}\,\mathit{eval}(E,V)}_i \big\langle E,\ \texttt{skip}\big\rangle}$$

$$\frac{\mathit{active}(i,c) \qquad \mathit{eval}(E,x_c) = \perp}{\big\langle E,\ c\texttt{!}V\big\rangle \xrightarrow{\tau}_i \big\langle E \oslash \{x_c \mapsto \mathit{eval}(E,V)\},\ c\texttt{!}V\big\rangle}$$

$$\frac{\mathit{active}(i,c) \qquad \mathit{eval}(E,x_c) \neq \perp}{\big\langle E,\ c\texttt{!}V\big\rangle \xrightarrow{c\texttt{!}\,\mathit{eval}(E,V)}_i \big\langle \mathit{reset}(E,x_c),\ \texttt{skip}\big\rangle}$$

Rules for Receptions. These rules are dual of those for emissions.

$$\frac{\mathit{passive}(i,c)}{\big\langle E,\ c\texttt{?}x\big\rangle \xrightarrow{c\texttt{?}\,\mathit{eval}(E,x_c)}_i \big\langle E \oslash \{x \mapsto \mathit{eval}(E,x_c)\},\ \texttt{skip}\big\rangle} \qquad (\mathsf{Recv_p})$$

$$\frac{\mathit{active}(i,c) \qquad \mathit{eval}(E,x_c) = \perp}{\big\langle E,\ c\texttt{?}x\big\rangle \xrightarrow{\tau}_i \big\langle E \oslash \{x_c \mapsto \mathit{ready}\},\ c\texttt{?}x\big\rangle}$$

$$\frac{\mathit{active}(i,c) \qquad \mathit{eval}(E,x_c) \neq \perp \qquad V \in \mathit{type}(c)}{\big\langle E,\ c\texttt{?}x\big\rangle \xrightarrow{c\texttt{?}V}_i \big\langle \mathit{reset}(E,x_c) \oslash \{x \mapsto V\},\ \texttt{skip}\big\rangle} \qquad (\mathsf{Recv_a})$$

Contrary to rule ($\mathsf{Recv_p}$), which uses the value of x_c as the received value, rule ($\mathsf{Recv_a}$) enumerates all possible values that might be received on channel c.

Rules for Sequential Composition. These rules are as usual.

$$\frac{\langle E,\ B_1\rangle \xrightarrow{L}_i \langle E',\ B_1'\rangle}{\langle E,\ B_1\,;B_2\rangle \xrightarrow{L}_i \langle E',\ B_1'\,;B_2\rangle} \qquad \frac{\langle E,B_1\rangle\surd \qquad \langle E,B_2\rangle \xrightarrow{L}_i \langle E',B_2'\rangle}{\langle E,\ B_1\,;B_2\rangle \xrightarrow{L}_i \langle E',B_2'\rangle}$$

Rules for Collateral Composition. These rules are as usual.

$$\frac{\langle E,\ B_1\rangle \xrightarrow{L}_i \langle E',\ B_1'\rangle}{\langle E,\ B_1\,,B_2\rangle \xrightarrow{L}_i \langle E',\ B_1'\,,B_2\rangle} \quad \frac{\langle E,\ B_2\rangle \xrightarrow{L}_i \langle E',\ B_2'\rangle}{\langle E,\ B_1\,,B_2\rangle \xrightarrow{L}_i \langle E',\ B_1\,,B_2'\rangle} \quad \frac{\langle E,B_1\rangle\surd \qquad \langle E,B_2\rangle\surd}{\langle E,\ B_1\,,B_2\rangle\surd}$$

Rules for Guarded Commands. The rules for guarded commands express that a branch whose guard is true can be selected. If the chosen branch ends with `break`, the guarded command terminates when the branch terminates; if it ends with `loop` the guarded command will be executed once more after executing the branch.

$$\frac{(\exists i \in \{1,\ldots,n\}) \qquad \mathit{eval}(E,G_i) = \mathit{true} \qquad T_i = \texttt{break} \qquad \langle E,B_i\rangle \xrightarrow{L}_i \langle E',B_i'\rangle}{\big\langle E,\ \texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}\big\rangle \xrightarrow{L}_i \big\langle E',\ B_i'\big\rangle}$$

$$\frac{(\exists i \in \{1,\ldots,n\}) \qquad \mathit{eval}(E,G_i) = \mathit{true} \qquad T_i = \texttt{loop} \qquad \langle E,B_i\rangle \xrightarrow{L}_i \langle E',B_i'\rangle}{\big\langle E,\ \texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}\big\rangle \xrightarrow{L}_i \big\langle E',\ B_i'\texttt{;}\ \texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}\big\rangle}$$

3.3 Semantics for Communicating Processes

The semantics of a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_1 \parallel \cdots \parallel \hat{B}_n\rangle$ is defined by the parallel composition of the LTSs $\langle \mathcal{S}_i, \mathcal{L}_i, \rightarrow_i, \langle E_i, \hat{B}_i\rangle\rangle$ produced for the individual processes $\hat{B}_i$ as defined in Section 3.2. This yields a new LTS $\langle \mathcal{S},\ \mathcal{L},\ \rightarrow, \langle E, \hat{B}_1, \ldots, \hat{B}_n\rangle\rangle$, where:

- The set of states $\mathcal{S}$ contains tuples $\langle E, B_1, \ldots, B_n\rangle$ of n processes B_1, ..., B_n and a global environment E on $\mathcal{X}^* = \bigcup_{i=1}^{n} \mathcal{X}_i^*$. E is the union of the local environments E_i on $\mathcal{X}_i^*$ of the processes $\hat{B}_i$. The sets $\mathcal{X}_i^*$ are disjoint for the sets $\mathcal{X}_i$ (local variables of $\hat{B}_i$), but for each binary channel c, the variable x_c occurs in $\mathcal{X}_i^*$ and $\mathcal{X}_j^*$ $(i \neq j)$; this is not a problem, since x_c is only modified by the process active for c.
- The set of labels $\mathcal{L}$ is the union of the sets of labels $\mathcal{L}_i$ minus labels corresponding to receptions on binary channels. We represent synchronised communications (i.e. an emission and a reception) using the same symbol "`!`" as for emissions (following the convention used in CADP).
- The transition relation "$\rightarrow$" is defined by the three SOS rules below.
- The initial state is $\langle E, \hat{B}_1, \ldots, \hat{B}_n\rangle$, with an empty initial environment E.

Let $\mathit{internal}(i, L)$ be the predicate that holds iff L is internal or a communication on a port (i.e. a unary channel open to the environment):

$$(\forall i \in \{1,\ldots,n\})\ (\forall L \in \mathcal{L})\ \mathit{internal}(i,L) \Longleftrightarrow \Big(L = \tau\ \vee\ \big((\exists c \in \mathcal{C})\ (\exists V \in \mathcal{V})\ (L = c!V\ \vee\ L = c?V)\ \wedge\ \mathit{port}(i,c)\big)\Big)$$

The first SOS rule describes the local — or internal — evolution of the i-th process B_i independently of the others. It models either an assignment to a variable, or the communication on a port c which is open to the environment and does not need to be synchronised with another process B_j ($i \neq j$).

$$\frac{(\exists i \in \{1,\ldots,n\}) \qquad \langle E, B_i\rangle \xrightarrow{L}_i \langle E', B_i'\rangle \qquad internal(i,L)}{\langle E, B_1, \ldots, B_i, \ldots, B_n\rangle \xrightarrow{L} \langle E', B_1, \ldots, B_i', \ldots, B_n\rangle}$$

The next rule describes the synchronisation between processes B_i and B_j communicating on channel c, B_i being the emitter and B_j the receiver.

$$\frac{(\exists i \in \{1,\ldots,n\})\ \langle E, B_i\rangle \xrightarrow{c!V}_i \langle E', B_i'\rangle \qquad (\exists j \in \{1,\ldots,n\})\ \langle E, B_j\rangle \xrightarrow{c?V}_i \langle E'', B_j'\rangle}{\langle E, B_1, \ldots, B_i, \ldots, B_j, \ldots, B_n\rangle \xrightarrow{c!V} \langle reset(E'', x_c), B_1, \ldots, B_i', \ldots, B_j', \ldots, B_n\rangle} \quad (\mathsf{Com})$$

Note that i and j in rule (Com) are different and uniquely defined, since the communication model is binary (one sender, one receiver for a given channel). Note also that, if E' and E differ in rule (Com), then the only possible modification (resetting x_c) is applied to E'' in the right hand side of the conclusion of rule (Com).

The rules presented so far are sufficient to define the semantics of (closed) systems without passive ports, i.e. unary channels for which no process $\hat{B}_i$ is active. The following rule completes the semantics by modelling the environment as an active process that communicates with each passive port c.

$$\frac{(\exists i \in \{1,\ldots,n\})\ passive(i,c)\ \ (\forall j \in \{1,\ldots,n\})\ \neg active(j,c)\ \ eval(E, x_c) = \bot\ \ V \in type(x_c)}{\langle E, B_1, \ldots, B_n\rangle \xrightarrow{\tau} \langle E \oslash \{x_c \mapsto V\}, B_1, \ldots, B_n\rangle} \quad (\mathsf{Env})$$

This rule is similar to those employed in the definition of semantics for asynchronous processes communicating via shared memory, as for instance concurrent constraint programming [dBP91] or concurrent declarative programming [Ser02, Table 5.3, page 142].

Example 1 *This example shows the necessity of rule* (Env). *Consider the following two processes* $B_1 =$ `@[`c_1 `#1 ⇒ (`c`!1, `c_1`?`$x`); loop]` *and* $B_2 =$ `@[`c_2 `#2 ⇒ (`c`!2, `c_2`?`$x`); loop]` *corresponding to the two branches of the arbiter of Section 2.3. Here,* c_1 *and* c_2 *are passive ports open to the environment. Without rule* (Env), *both* B_1 *and* B_2 *would be equivalent to the deadlock process* `nil`. *However, while "*$B_1 \parallel$ *client-2" is equivalent to* `nil`, *"*$B_2 \parallel$ *client-2" is not (the corresponding* LTS, *shown in Figure 2, has 8 states and 12 transitions). Rule* (Env) *solves this issue by giving a different semantics to* B_1 *and* B_2.

Example 2 *Figure 3 gives the* LTS *generated for the arbiter of Section 2.3. To keep the size of Figure 3 as small as possible, we minimised the* LTS *with respect to strong bisimulation (merging states that differ only in the value of variable* x *when* x *is no longer used). This is similar to the state reduction approach for process algebra described in* [GS04].

3.4 Comparison with the Existing Petri Net Translation

In this section, we compare our SOS semantics with the "implicit" semantics proposed for CHP in [RY04] by a translation of CHP into Petri nets. [RY04] only handles a subset of CHP

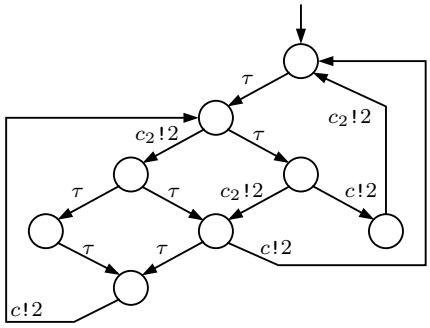


Figure 2: LTS for "$B_2 \parallel$ client-2" of Example 1

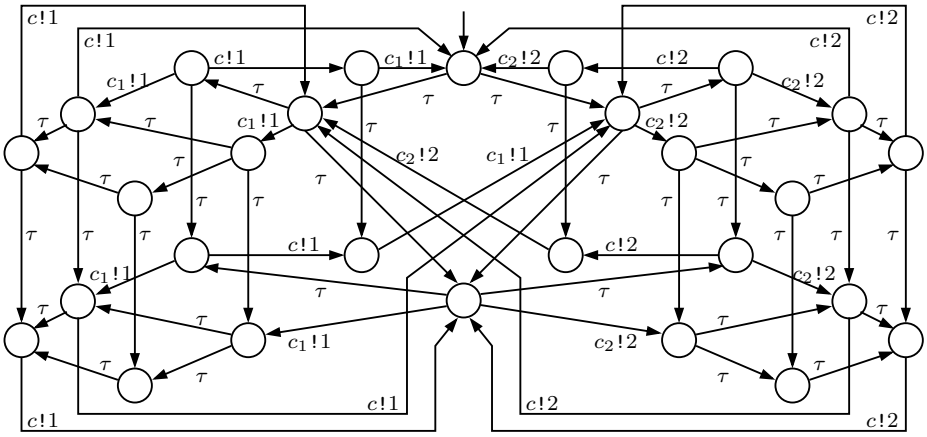


Figure 3: LTS for the arbiter example

that, compared to full CHP presented in Section 2, is restricted in two ways: it allows only pure synchronisations (instead of value-passing communications) and forbids ports open to the environment. By handling full CHP, our semantics allows to describe circuits with inputs and outputs properly.

Translation of CHP to Petri Nets. Similar to our SOS semantics, [RY04] defines the translation of a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_1 \parallel \cdots \parallel \hat{B}_n \rangle$ into Petri nets in two successive steps:

- In a first step the Petri nets corresponding to the processes $\hat{B}_i$ are constructed separately following the patterns sketched in [RY04]. Petri net places may be labelled with assignments, emissions, and receptions. Petri net transitions may be labelled with the guards of CHP guarded commands. To fire a transition, its input places must contain a token and the guard (if any) must be true.
- In a second step, the separate Petri nets are merged into one global Petri net. To model synchronisation on channels, [RY04] gives two different translations, depending on the handshake protocol (2- or 4-phase) used for the implementation. In both cases,

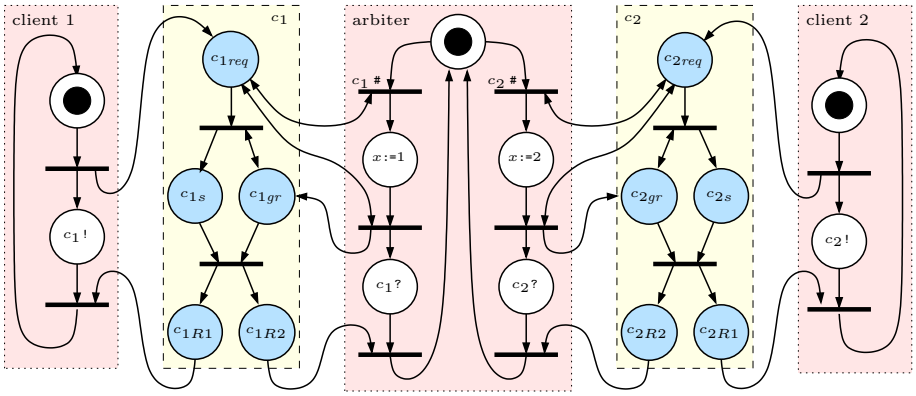


Figure 4: Petri net for the arbiter example

channels are modelled by additional places and transitions that encode the chosen handshake protocol. Notice that for each channel c the places labelled "c!" and "c?" are kept separate, i.e. there is no transition merging as in [GS90].

The generated Petri net is one-safe, i.e. in every reachable marking, each place contains at most one token.

Example 3 *Consider the* CHP *description* $\langle \{c_1, c_2\},\ \{x\},\ \textit{client-1} \parallel \textit{client-2} \parallel \textit{arbiter} \rangle$, *where channels* c_1 *and* c_2 *have an active emitter and a passive receiver, where variable* x *is local to the arbiter, and where the three processes are defined as follows:*

client-1: `@[`$\textit{true} \Rightarrow c_1$`!; loop]`
client-2: `@[`$\textit{true} \Rightarrow c_2$`!; loop]`
arbiter: `@[`c_1 `#1` $\Rightarrow x$`:=1;` c_1`?; loop  `c_2 `#2` $\Rightarrow x$`:=2;` c_2`?; loop]`

This example is an adaptation of the arbiter of Section 2.3 in order to meet the restrictions of [RY04]. *The corresponding Petri net for a 4-phase protocol is (adapted from* [RY04, Figure 11]*) shown in Figure 4. Places are represented by circles, and transitions by thick lines. Whenever a place is both an input and an output place of some transition, we use a double-headed arrow (as for places labelled* c_{1req}, c_{1gr}, c_{2req}, *and* c_{2gr}*). The Petri nets corresponding to the three processes are framed in dotted boxes. The places modelling the channels* c_1 *and* c_2 *are framed in dashed boxes.*

Relation between SOS and Petri Net Semantics. In order to relate the Petri nets proposed in [RY04] with the LTSs produced by our semantics, one needs to generate the LTSs corresponding to the Petri nets. This is not immediate, since the Petri nets of [RY04] have a different behaviour than ordinary Petri nets. For instance, if two places with action (e.g. "c_1!" and "c_2!" in Figure 4) have a token, then interleaved transitions have to be created for these actions. From [RY04] and following discussions with the first author of [RY04], we conjecture that these LTSs can be obtained by the following two steps.

- First, the Petri net model of [RY04] with actions attached to places needs to be transformed into a more standard model where actions are attached to transitions.

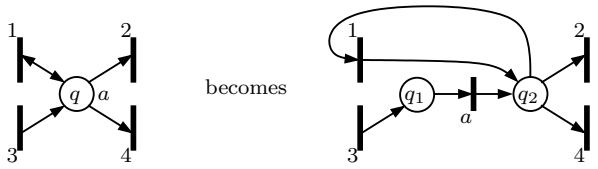


Figure 5: Duplication of Petri net places

As shown in Figure 5, each Petri net place q labelled with an action a (i.e. emission, reception, or assignment) is replaced by two places q_1 and q_2 and a new Petri net transition labelled with action a. Place q is replaced by q_1 (respectively q_2) in the sets of output (respectively input) places of all transitions. In the case a transition t has q both as an input and an output place (i.e. t corresponds to a double-headed arrow), q is replaced by q_2 in the sets of input and output places of t.

- Then, we compute the LTS for the modified Petri net by applying the standard marking graph construction algorithm. We label the transitions of the generated LTS with emissions and receptions labelling Petri net transitions. If a Petri net transition is not labelled with an emission or a reception, the corresponding LTS transition is labelled with τ.

We can now compare the LTS_{SOS} obtained by our SOS semantics and LTS_{PN} obtained after translation of CHP into Petri nets, transformation, and marking graph construction. Given that [RY04] does not deal with value-passing communications and open systems, this is only possible for closed systems with pure synchronisations.

A first remark is that the places and transitions added to the Petri nets for the communication channels introduce τ transitions in LTS_{PN} that might not have a counterpart in LTS_{SOS}. Thus, LTS_{SOS} and LTS_{PN} are not strongly equivalent. A second remark is that the sets of labels of LTS_{SOS} and LTS_{PN} might be different. On the one hand, LTS_{PN} might contain both, "c!" and "c?" as labels, since the places labelled "c!" and "c?" are kept separate in the Petri nets of [RY04]. On the other hand, for closed systems LTS_{SOS} does not contain labels of the form "c?". Establishing an equivalence relation between LTS_{SOS} and LTS_{PN} would probably require to rename into τ all labels of LTS_{PN} corresponding to communications by active processes and to replace all remaining "?" by "!"; we conjecture that after these transformations, LTS_{SOS} and LTS_{PN} are equivalent with respect to branching equivalence [vGW89], but this remains to be proved.

4 Principles of a Translation from CHP to LOTOS

In order to check the correctness of asynchronous circuit designs (e.g. absence of deadlocks), our approach is to translate CHP into LOTOS so that existing tools (namely, the CADP verification toolbox [GLM02b]) can be applied. A tutorial of the ISO standard LOTOS [ISO89] can be found in [BB88]. We highlight first the main features of the translation of CHP into LOTOS:

- CHP types (natural numbers, Booleans, bit vectors, etc.) are translated into LOTOS sorts (i.e. algebraic data types).

- CHP functions are translated into LOTOS operations, the semantics of which is defined using algebraic equations.
- A CHP channel c is translated into a LOTOS gate with the same name c.
- A CHP variable x is translated into one or more LOTOS variables (i.e. *value identifiers* in the LOTOS standard terminology) with the same name and the same type as x. Several LOTOS variables might be required since LOTOS variables are single-assignment, whereas CHP variables are multiple-assignment.
- Sequential composition "`;`" in CHP is symmetric, whereas LOTOS has two different operators for sequential composition: an asymmetric action prefix "`;`" and a symmetric sequential composition "`>>`". Variables assigned on the left hand side of a CHP "`;`" can be used on the right hand side, whereas variables assigned on the left hand side of a LOTOS "`>>`" must be explicitly listed (in an `accept` clause) to be used in the right hand side. Furthermore, "`>>`" creates an internal τ transition, contrary to the "`;`" operator of both CHP and LOTOS. There are two options when translating CHP to LOTOS. A simple approach is to use only "`>>`". A better approach is to use the LOTOS "`;`" whenever possible, and "`>>`" only when needed. In this report, we adopt the second approach which generates better LOTOS code at the expense of a more involved translation.
- CHP has a neutral element (`skip`) for its sequential composition, whereas LOTOS lacks neutral elements both for "`;`" (which is asymmetric) and for "`>>`" (which creates a τ transition).
- CHP has a loop operator, whereas LOTOS does not; CHP loops have to be translated into recursive processes in LOTOS.
- CHP guards are either Boolean expressions or probes, whereas LOTOS guards are Boolean expressions only.

4.1 Principles of Translating a Single Process

The translation of a CHP process $\hat{B}_i$ is described by the recursive function $c2l_i(B, D, U, \Delta)$ with four parameters: B is a CHP process to translate and D, U, and Δ are alphabetically ordered sets of variables necessary to compute the variables to explicitly pass over LOTOS sequential compositions "`>>`". Intuitively, D is the set of variables that have a defined value before execution of B, U is the set of variables used after execution of B, and $\Delta \subseteq D$ is an auxiliary set of defined variables used to translate collateral compositions.

Data-flow Analysis. We introduce the following data-flow sets inspired from [GS04, Section 3]. Let $def(B)$ be the set of variables defined by process B:

$$
\begin{aligned}
&def(\texttt{nil}) = \emptyset \\
&def(\texttt{skip}) = \emptyset \\
&def(c\texttt{!}V) = \emptyset \\
&def(x\texttt{:=}V) = \{x\} \\
&def(c\texttt{?}x) = \{x\} \\
&def(B_1\texttt{;}B_2) = def(B_1) \cup def(B_2) \\
&def(B_1\texttt{,}B_2) = def(B_1) \cup def(B_2) \\
&def(\texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}) = \bigcup_{i=1}^{n} def(B_i)
\end{aligned}
$$

Let $use_v(V)$ be the set of variables used by value expression V:

$$
\begin{aligned}
&use_v(x) = \{x\} \\
&use_v\big(f(V_1, \ldots, V_n)\big) = \textstyle\bigcup_{i=1}^{n} use_v(V_i)
\end{aligned}
$$

Let $use(B)$ be the set of variables used by process B before they are defined:

$$
\begin{aligned}
&use(\texttt{nil}) = \emptyset \\
&use(\texttt{skip}) = \emptyset \\
&use(x\texttt{:=}V) = use_v(V) \\
&use(c\texttt{!}V) = use_v(V) \\
&use(c\texttt{?}x) = \emptyset \\
&use(B_1\texttt{;}B_2) = use(B_1) \cup \big(use(B_2) \setminus def(B_1)\big) \\
&use(B_1\texttt{,}B_2) = use(B_1) \cup use(B_2) \\
&use(\texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}) = \bigcup_{i=1}^{n} \big(use(G_i) \cup use(B_i)\big)
\end{aligned}
$$

Functionalities. The functionality of a CHP process B is given by the function $func(B, D, U)$, where D and U are two alphabetically ordered sets of variables with the same intuition as for $c2l_i$. A functionality is either `noexit` or `exit`(X), X being a possibly empty alphabetically ordered set of variables.

$$
\begin{aligned}
&func(\texttt{nil}, D, U) = \texttt{noexit} \\
&func(\texttt{skip}, D, U) = \texttt{exit}(D \cap U) \\
&func(c\texttt{!}V, D, U) = \texttt{exit}(D \cap U) \\
&func(x\texttt{:=}V, D, U) = \texttt{exit}\big((D \cup \{x\}) \cap U\big) \\
&func(c\texttt{?}x, D, U) = \texttt{exit}\big((D \cup \{x\}) \cap U\big) \\
&func(B_1\texttt{;}B_2, D, U) = \ \text{if } func(B_1, D, U) = \texttt{noexit} \ \vee\ func(B_2, D, U) = \texttt{noexit} \text{ then} \\
&\qquad\qquad \texttt{noexit} \\
&\qquad\quad \text{else} \\
&\qquad\qquad \texttt{exit}\big((D \cup def(B_1) \cup def(B_2)) \cap U\big) \\
&\qquad\quad \text{end if}
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{func}(B_1\,\texttt{,}\,B_2, D, U) = \ \text{if } \mathit{func}(B_1, D, U) = \texttt{noexit} \ \vee \ \mathit{func}(B_2, D, U) = \texttt{noexit} \text{ then} \\
\qquad\qquad\qquad\qquad\quad \texttt{noexit} \\
\qquad\qquad\qquad\qquad \text{else} \\
\qquad\qquad\qquad\qquad\quad \texttt{exit}\big((D \cup \mathit{def}(B_1) \cup \mathit{def}(B_2)) \cap U\big) \\
\qquad\qquad\qquad\qquad \text{end if} \\
\mathit{func}(\texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}, D, U) = \\
\qquad \text{if } (\forall i \in \{1, \ldots, n\})\ \mathit{func}(B_i, D, U) = \texttt{noexit} \ \vee \ T_i = \texttt{loop} \text{ then} \\
\qquad\quad \texttt{noexit} \\
\qquad \text{else} \\
\qquad\quad \texttt{exit}\Big(\big(D \cup \bigcup_{i=1}^{n} \mathit{def}(B_i)\big) \cap U\Big) \\
\qquad \text{end if}
\end{array}
$$

Using *func*, let $\mathit{inf}(B)$ be the predicate that holds iff $\mathit{func}(B, \emptyset, \emptyset) = \texttt{noexit}$.

Preliminary Transformations. We first simplify the CHP processes by applying the following transformations successively:

- All occurrences of `skip` are removed wherever possible, based on the facts that (1) `skip` is neutral element for sequential and collateral composition, (2) any branch "$G \Rightarrow$ `skip; loop`" of a guarded command can be removed, and (3) any $\hat{B}_i$ equal to `skip` can be removed from $\hat{B}_1 \parallel \cdots \parallel \hat{B}_n$. After these transformations, `skip` may occur only in branches "$G \Rightarrow$ `skip; break`" of guarded commands.
- The abstract syntax tree of each CHP process is reorganised so as to be right bracketed (based on the associativity of CHP sequential composition). After transformation, each sequence "B_1`;` B_2`;` B_3" is bracketed as "B_1`;` (B_2`;` B_3)".
- If the rightmost process B_n of a maximal sequence "B_1`;` $\ldots$ `;`B_n" ($n \geq 1$) is of the form "x`:=`V", "c`!`V", or "c`?`x" (and not followed by `break` or `loop`), a final `skip` is added, leading to the sequence "B_1`;` $\ldots$ `;`B_n`; skip`".
- For each process of the form "B_1`;`B_2" such that $\mathit{inf}(B_1)$, B_2 will never be executed and can be removed. Similarly, in each process of the form "B_1`,`B_2" such that $\mathit{inf}(B_1)$ is the negation of $\mathit{inf}(B_2)$, we replace the process B_i ($i \in \{1, 2\}$) such that $\mathit{inf}(B_i)$ does not hold, by "B_i`; nil`". Also, if $\neg\mathit{inf}(\hat{B}_i)$, then $\hat{B}_i$ is replaced by "$\hat{B}_i$`;nil`". These transformations are needed to obey the static check of functionalities in LOTOS.

After these transformations, all assignments and all communications (emissions and receptions) are used in prefix-style, i.e. they occur only in processes of one of the forms "x`:=`V`;` B", "c`!`V`;` B", and "c`?`x`;` B". Formally, after these transformations, CHP processes are described by the following grammar:

$$
\begin{array}{l}
B ::= \texttt{nil} \ \mid \ \texttt{skip} \ \mid \ A\texttt{;}B \ \mid \ B_1\texttt{,}B_2 \ \mid \ \texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]} \\
A ::= x\texttt{:=}V \ \mid \ c\texttt{!}V \ \mid \ c\texttt{?}x \ \mid \ B_1\texttt{,}B_2 \ \mid \ \texttt{@[}G_1 \Rightarrow B_1\texttt{;}\ T_1\ \ldots\ G_n \Rightarrow B_n\texttt{;}\ T_n\texttt{]}
\end{array}
$$

where G, T, and V are defined as in Section 2. In the sequel, we use this new grammar for the definition of the translation function *c2l*.

Translation of `nil` and `skip`. `nil` is translated into `stop`. After the preliminary transformations `skip` occurs only in a guarded command as a branch "$G \Rightarrow$ `skip; break`" (this case is handled below with guarded commands) or at the end of a sequence; in this case:

$$c2l_i(\texttt{skip}, D, U, \Delta) = \texttt{exit}(\xi_1, \ldots, \xi_n)$$

where the ξ_i are defined as follows. Let $\{x_1, \ldots, x_n\} = D \cap U$. Then $(\forall i \in \{1, \ldots, n\})$ $\xi_i = x_i$ if $x_i \in \Delta$ or $\xi_i =$ "`any` $type(x_i)$" otherwise.

Translation of "x`:=`V`;` B". An assignment to a variable x of type S is translated (generating an internal transition as in our SOS semantics and [RY04]) into:

$$c2l_i(\text{“}x\texttt{:=}V\texttt{;}\,B\text{”}, D, U, \Delta) = \texttt{let}\ \ x\texttt{:}S\texttt{ = }V\ \texttt{in}\ \ \tau\texttt{;}\ \ c2l_i\big(B, D \cup \{x\}, U, \Delta \cup \{x\}\big)$$

Translation of Guards. Boolean expressions "$V \Rightarrow$" are directly translated into "V `->`". To model the CHP probe operator for a channel c, we introduce an additional synchronisation on the corresponding LOTOS gate c. Probes are distinguished from actual communications by an additional offer "`!`*Probe*", where *Probe* is a special constant belonging to an enumerated type with a single value. This translation is based on the value-matching feature of LOTOS synchronisation, which ensures that two offers "`!`*Probe*" will synchronise.

$$c2l_g(c\,\texttt{\#}) = c\,\texttt{!}\,Probe$$
$$c2l_g(c\,\texttt{\#}V) = c\,\texttt{!}\,Probe\,\texttt{!}\,V$$

Translation of "c`!`V`;` B" and "c`?`x`;` B". Translation of an emission or a reception on a channel c of type $S = type(c)$ depends whether process $\hat{B}_i$ is active or passive for c.

- The translation is straightforward if $passive(i, c)$ holds:

$$c2l_i(\text{“}c\texttt{!}V\texttt{;}\,B\text{”}, D, U, \Delta) = c\texttt{!}V\texttt{;}\ \ c2l_i(B, D, U, \Delta)$$
$$c2l_i(\text{“}c\texttt{?}x\texttt{;}\,B\text{”}, D, U, \Delta) = c\texttt{?}x\texttt{:}S\texttt{;}\ \ c2l_i(B, D \cup \{x\}, U, \Delta \cup \{x\})$$

- If $active(i, c)$ holds, the translation is more involved because the active process $\hat{B}_i$ needs to allow its passive partner to probe channel c an arbitrary number of times. After a synchronisation labelled with "`!`*Probe*", $\hat{B}_i$ can only perform further synchronisations labelled with "`!`*Probe*", until the communication is completed. Therefore, for every occurrence of an active emission or an active reception on channel c in $\hat{B}_i$, we define an auxiliary LOTOS process *probed_c*, the definition of which depends whether c is used for emission or reception.

  - An active emission "c`!`V`;` B" translates as follows:

$$c2l_i(\text{“}c\texttt{!}V\texttt{;}\,B\text{”}, D, U, \Delta) =$$
$$\quad \tau\texttt{;}\ \ probed\_c\texttt{[}c\texttt{]}(V, x_1, \ldots, x_n)\ \texttt{>>}\ \texttt{accept}\ \ x_1\texttt{:}S_1, \ldots, x_n\texttt{:}S_n\ \ \texttt{in}$$
$$\quad c2l_i(B, D', U, \Delta \cap D')$$

    where $D' = D \cap (use(B) \cup U)$, $\{x_1, \ldots, x_n\} = D'$, and $(\forall i \in \{1, \ldots, n\})$ $S_i = type(x_i)$. Process *probed_c* is defined by:

```
process probed_c[c](x:S, x1:S1, ..., xn:Sn): exit(S1, ..., Sn) :=
  c!x; exit(x1, ..., xn)  []  c!Probe!x; probed_c[c](x, x1, ..., xn)
endproc
```

– An active reception "$c?x\,;\,B$" translates as follows:

$$\begin{array}{l} c2l_i(\text{“}c?x\,;\,B\text{”}, D, U, \Delta) = \\ \quad \tau\,;\ \mathit{probed\_c}[c](x_1, \ldots, x_n) \texttt{ >> accept } x_1 : S_1, \ldots, x_n : S_n \texttt{ in} \\ \quad c2l_i(B, D', U, \Delta \cap D') \end{array}$$

where $D' = (D \cup \{x\}) \cap (\mathit{use}(B) \cup U)$, $\{x_1, \ldots, x_n\} = D'$, and $(\forall i \in \{1, \ldots, n\})$ $S_i = \mathit{type}(x_i)$. Process $\mathit{probed\_c}$ is defined by:

$$\begin{array}{l} \texttt{process } \mathit{probed\_c}[c](x_1 : S_1, \ldots, x_n : S_n)\texttt{: } \texttt{ exit}(S_1, \ldots, S_n) \texttt{ :=} \\ \quad c?x : S\texttt{; exit}(x_1, \ldots, x_n) \texttt{ [] } c\texttt{!}\mathit{Probe}\texttt{; } \mathit{probed\_c}[c](x_1, \ldots, x_n) \\ \texttt{endproc} \end{array}$$

If $x \in \{x_1, \ldots, x_n\}$ and $x \notin D$, then x is removed from the parameters of process $\mathit{probed\_c}$.

In both cases, the τ action preceding the call to $\mathit{probed\_c}$ is created by the assignment to x_c (cf. Section 3) and models the asymmetry of Chp communications, i.e. the fact that the active process chooses first. Notice that redundant process definitions can be avoided by defining $\mathit{probed\_c}$ only for all relevant subsets of $\{x_1, \ldots, x_n\}$; an approximation of these subsets can be computed by data-flow analysis.

Contrary to our approach, the translation from Balsa into Csp sketched in [WKTZ04] uses pairs of actions with different names for representing (passive) input enclosures; the active process is translated accordingly. Our approach has the advantage that the translation of an active process is independent of the fact that the passive process probes or not.

Translation of "$A\,;B$". This translation rule applies only if A is a collateral composition or a guarded command; all other cases have been handled before.

$$\begin{array}{l} c2l_i(\text{“}A\,;B\text{”}, D, U, \Delta) = \\ \qquad c2l_i(A, \Delta', U', \Delta') \texttt{ >> accept } x_1 : S_1, \ldots, x_n : S_n \texttt{ in } c2l_i(B, D', U, \Delta') \end{array}$$

where $U' = \mathit{use}(B) \cup \big(U \setminus \mathit{def}(B)\big)$, $D' = \big(D \cup \mathit{def}(A)\big) \cap U'$, $\Delta' = \big(\Delta \cup \mathit{def}(A)\big) \cap U'$, and $\{x_1, \ldots, x_n\} = \Delta'$.

Translation of "$B_1\,,B_2$". A collateral composition is translated as follows:

$$c2l_i(\text{“}B_1\,,B_2\text{”}, D, U, \Delta) = c2l_i(B_1, D', U, \Delta_1) \texttt{ ||| } c2l_i(B_2, D', U, \Delta_2)$$

where $D' = D \cup \mathit{def}(B_1) \cup \mathit{def}(B_2)$, $\Delta_1 = D' \setminus \mathit{def}(B_2)$, and $\Delta_2 = D' \setminus \mathit{def}(B_1)$.

Translation of Guarded Commands. Every guarded command B = "$\texttt{@[}G_1 \Rightarrow B_1\texttt{; }T_1\ \ldots\ G_n \Rightarrow B_n\texttt{; }T_n\texttt{]}$" is translated into a call to a process P_B

$$c2l_i(\texttt{@[}G_1 \Rightarrow B_1\texttt{; }T_1\ \ldots\ G_n \Rightarrow B_n\texttt{; }T_n\texttt{]}, D, U, \Delta) = P_B\texttt{[}\mathcal{C}_i\texttt{]}(x_1, \ldots, x_n)$$

where $\{x_1, \ldots, x_n\} = D$. The auxiliary process P_B is defined by:

$$\texttt{process}\ P_B\texttt{[}\mathcal{C}_i\texttt{]}(x_1\texttt{:}S_1,\ldots,x_n\texttt{:}S_n)\texttt{:}\ F$$
$$\quad c2l_g(G_1)\ c2l_i(B_1,D,U,\Delta)\ \texttt{>>}\ c2l_t(T_1,B_1,D,U,\Delta)\quad \texttt{[]}\cdots\texttt{[]}$$
$$\quad c2l_g(G_n)\ c2l_i(B_n,D,U,\Delta)\ \texttt{>>}\ c2l_t(T_n,B_n,D,U,\Delta)$$
$$\texttt{endproc}$$

$$c2l_t(\texttt{loop},B,D,U,\Delta) = \texttt{accept}\ x_1\texttt{:}S_1,\ldots,x_n\texttt{:}S_n\ \texttt{in}\ P_B\texttt{[}\mathcal{C}_i\texttt{]}(x_1,\ldots,x_n)$$
$$c2l_t(\texttt{break},B,D,U,\Delta) =$$
$$\quad \texttt{accept}\ x'_1\texttt{:}S_1,\ldots,x'_m\texttt{:}S_m\ \texttt{in}\ c2l_i\big(\texttt{skip},(D\cup def(B))\cap U,U,\Delta\big)$$

with $\{x'_1,\ldots,x'_m\} = \big(D \cup def(B)\big) \cap U$. If $inf(B)$ then $F = \texttt{noexit}$; otherwise $func(B,D,U) = \texttt{exit}(\{x''_1,\ldots,x''_k\})$ and $F = \texttt{exit}(type(x''_1),\ldots,type(x''_k))$.

If a guarded command is the left hand side of a sequential composition "$\texttt{@[}\ldots\texttt{]};\ B'$", we generate a second auxiliary process $P_{B'}$ (for B'); each `break` is translated into a call to $P_{B'}$ and the functionality F is computed with respect to B'. This avoids the introduction of a τ transition due to the `exit` (generated by the translation of `break`). For each B_i such that $inf(B_i)$, T_i is not translated at all.

4.2 Principles of Translating Several Concurrent Processes

The parallel composition "$\|$" of CHP is translated into the LOTOS operator "$\texttt{|[}\cdots\texttt{]|}$". In CHP, processes synchronise implicitly on channels with the same name, whereas in LOTOS the set of gates for synchronisation has to be stated explicitly. Although such a translation would not always be possible in the general case [GS99], it works for CHP descriptions, since the channels have pairwise distinct names.

The translation of a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_1 \parallel \cdots \parallel \hat{B}_n \rangle$ is defined recursively:

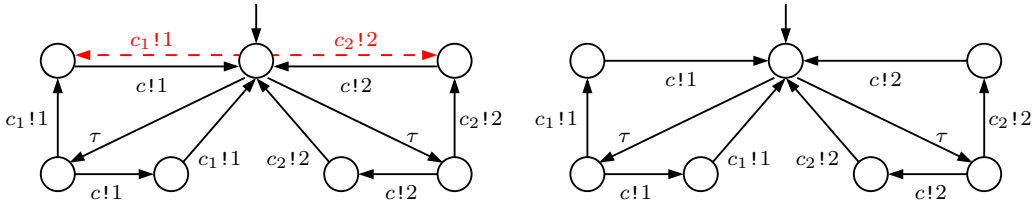
$$c2l(\text{“}\hat{B}_1 \parallel \cdots \parallel \hat{B}_n\text{”},\ \mathcal{C}) =$$
$$\quad \text{if } n = 1 \text{ then}$$
$$\quad\quad c2l_1(\hat{B}_1,\emptyset,\emptyset,\emptyset)$$
$$\quad \text{else}$$
$$\quad\quad c2l_1(\hat{B}_1,\emptyset,\emptyset,\emptyset)\ \texttt{|[}chan(\hat{B}_1)\cap\mathcal{C}\texttt{]|}\ c2l\big(\text{“}\hat{B}_2 \parallel \cdots \parallel \hat{B}_n\text{”},\ \mathcal{C}\setminus chan(\hat{B}_1)\big)$$
$$\quad \text{end if}$$

where $chan(\hat{B})$ stands for the set of binary channels occurring in process $\hat{B}$.

4.3 Example of the Arbiter

The LOTOS behaviour obtained as translation of the arbiter described in Section 2.3 is defined by the following LOTOS code:

$$client\text{-}1\texttt{[}c_1\texttt{]}\ \ \texttt{|[}c_1\texttt{]|}\ \ \big(client\text{-}2\texttt{[}c_2\texttt{]}\ \ \texttt{|[}c_2\texttt{]|}\ \ arbiter\texttt{[}c,c_1,c_2\texttt{]}\big)$$



(a) SOS semantics (b) LOTOS translation

Figure 6: LTSs of the arbiter example (minimised with respect to branching equivalence)

where

```
process client-1[c1] :  noexit :=
    τ; probed_c1[c1](1) >> client-1[c1]
endproc
process client-2[c2] :  noexit :=
    τ; probed_c2[c2](2) >> client-2[c2]
endproc
process arbiter[c, c1, c2] :  noexit :=
    (c1 !Probe !1; (τ; probed_c[c](1) ||| c1?x:S; exit) >> arbiter[c, c1, c2]) []
    (c2 !Probe !2; (τ; probed_c[c](2) ||| c2?x:S; exit) >> arbiter[c, c1, c2])
endproc
```

where the processes $probed\_c_{(j)}$ are defined as described in Section 4.1.

The LTS corresponding to this LOTOS specification has 82 states and 212 transitions. The LTS obtained after hiding all labels with "!*Probe*" offers was found (by the BISIMULATOR tool [BDJM05] of CADP) to be observationally [Mil80] (but not branching [vGW89]) equivalent to the one presented in Figure 3.

The reason why our translation does not preserve branching equivalence is that a probe is translated into a τ transition that is not present in the SOS semantics of Section 3. For instance, the LTS of Figure 3 contains a state (lower middle of the graph) with two outgoing transitions labelled "$c_1!1$" and "$c_2!2$", but no such state exists in the LTS obtained after our LOTOS translation, since both branches of the "`[]`" choice in the arbiter start with a τ transition corresponding to a probe.

To illustrate further, the LTSs depicted in Figure 6(a) (respectively Figure 6(b)) corresponds to the LTS obtained for the arbiter example by the SOS semantics (respectively the LOTOS translation) after minimisation with respect to branching equivalence. The transitions that are present in Figure 6(a), but not in Figure 6(b) are represented by dashed arrows. Notice that the two LTSs of Figure (6) are observationally equivalent.

4.4 Application: An Asynchronous Implementation of the DES

We developed a prototype CHP to LOTOS translator, called `chp2lotos`, using the SYNTAX and LOTOS NT compiler construction technologies [GLM02a]. So far, `chp2lotos` consists of about 2,000 lines of SYNTAX, 6,000 lines of LOTOS NT, and 500 lines of C.

We have experimented `chp2lotos` on a case study tackled in [BBM+03], namely a CHP description (1,600 lines) of an asynchronous implementation of the DES (*Data Encryption*

Standard) [NIS99], from which `chp2lotos` produced about 1,600 lines of LOTOS. Since this case study contains many concurrent processes, direct generation of the LTS failed due to lack of memory (after 70 minutes, the generated LTS had more than 17 million states and 139 million transitions). However, using the compositional verification techniques (decomposing, minimising, and recomposing processes) [Lan02] of the CADP toolbox we generated an equivalent, but smaller LTS (50,956 states and 228,136 transitions) in 8 minutes on a SunBlade 100 (500 Mhz Ultra Sparc II processor, 1.5 GB of RAM). Compared to [BBM+03][4], compositional techniques improves verification performance.

5 Concluding Remarks

In this report, we gave an SOS semantics for the hardware process algebra CHP with value-passing communication and ports open to the environment. Our semantics clarifies the definition of the probe operator. We investigated the relation of our semantics with existing semantics based on a translation of CHP into Petri nets. Based on our semantics, we outlined a translation of CHP into LOTOS, in order to allow the reuse of existing formal verification tools, such as CADP, for the analysis and validation of asynchronous hardware. Finally, we reported on a first experiment with a prototype translator.

As regards future work, we are currently validating and extending our prototype with the aim of an integration of formal verification in the design process by a tight connection between synthesis and verification tools, in particular TAST and CADP. Furthermore, it would be interesting to characterise precisely the weak equivalence preserved by our translation from CHP into LOTOS. We conjecture that observational equivalence is preserved, and that if the CHP description does not contain any probe, branching equivalence is also preserved.

Acknowledgements.

We are grateful to Edith Beigné, François Bertrand, Dominique Borrione, Marc Renaudin, and Pascal Vivet for interesting discussions on CHP and TAST, in particular on the semantics of the probe operator. We are indebted to Hubert Garavel for his significant contribution to the final version.

References

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[BBM+03] Dominique Borrione, Menouer Boubekeur, Laurent Mounier, Marc Renaudin, and Antoine Sirianni. Validation of Asynchronous Circuit Specifications using IF/CADP. In Manfred Glesner, Ricardo Augusto da Luz Reis, Hans Eveking, Vincent John Mooney, Leandro Soares Indrusiak, and Peter Zipf, editors, *Proceedings of the International Conference on Very Large Scale Integration of*

[4] [BBM+03] managed to generate an LTS directly (5.3 million states, 30 million transitions) in 65 minutes after replacing collateral compositions by sequential compositions in some well-chosen processes of the CHP description (according to the second author of [BBM+03]). When applying the same transformations on a LOTOS specification, we generated an equivalent LTS of the same size in 15 minutes.

System-on-Chip VLSI-SoC 2003 (Darmstadt, Germany), pages 86–91, Darmstadt, December 2003.

[BCV+05] Edith Beigné, Fabien Clermidy, Pascal Vivet, Alain Clouard, and Marc Renaudin. An Asynchronous NOC Architecture Providing Low Latency Service and Its Multi-Level Design Framework. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC'05 (New York, USA)*, pages 54–63. IEEE Computer Society Press, March 2005.

[BDJM05] Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[dBP91] Frank S. de Boer and Catuscia Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT'91, Volume 1, Colloquium on Trees in Algebra and Programming CAAP'91 (Brighton, United Kingdom)*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer Verlag, April 1991.

[EB02] Doug Edwards and Andrew Bardsley. Balsa: An Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.

[Fok00] Wan Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer Verlag, 2000.

[GLM02a] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction CC 2002 (Grenoble, France)*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer Verlag, April 2002.

[GLM02b] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[GS99] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[GS04] Hubert Garavel and Wendelin Serwe. State Space Reduction for Process Algebra Specifications. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology AMAST'2004 (Stirling, Scotland, UK)*, volume 3116 of *Lecture Notes in Computer Science*, pages 164–180. Springer Verlag, July 2004.

[Hau95] Scott Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[ISO89] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[KP01] Joep L. W. Kessels and Ad M. G. Peeters. The Tangram Framework (Embedded Tutorial): Asynchronous Circuits for Low Power. In *Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC 2001 (Yokohama, Japan)*, pages 255–260. ACM, 2001.

[Lan02] Frédéric Lang. Compositional Verification using SVL Scripts. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*, pages 465–469. Springer Verlag, April 2002.

[Mar85] Alain J. Martin. The Probe: An Addition to Communication Primitives. *Information Processing Letters*, 20(3):125–130, April 1985.

[Mar86] Alain J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(4):226–234, 1986.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[NIS99] NIST. Data Encryption Standard (DES). Federal Information Processing Standards FIPS PUB 46-3, National Institute of Standards and Technology, October 25 1999.

[Ren05] Marc Renaudin. *TAST Compiler and TAST_CHP Language, Version 0.6*. TIMA Laboratory, CIS Group, 2005.

[RY04] Marc Renaudin and Alex Yakovlev. From Hardware Processes to Asynchronous Circuits via Petri Nets: an Application to Arbiter Design. In *Proceedings of the Workshop on Token Based Computing TOBACO'04 (Bologna, Italy)*, June 2004.

[Ser02] Wendelin Serwe. *On Concurrent Functional-Logic Programming*. Thèse de doctorat, Institut National Polytechnique de Grenoble, March 2002.

[vB93] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[WKTZ04] X. Wang, Marta Kwiatkowska, G. Theodoropoulos, and Q. Zhang. Towards a Unifying CSP approach for Hierarchical Verification of Asynchronous Hardware. In M. R. A. Huth, editor, *Proceedings of the 4th International Workshop on Automated Verification of Critical Systems AVoCS'04 (London, UK)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 231–246, 2004.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopˆole de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399